PASCAL-80

Phelps Gates

NEW CLASSICS SOFTWARE
239 FOX HILL ROAD•DENVILLE NEW JERSEY 07834•TELEPHONE 201-625-8838

Table of Contents

What is **Pascal 80?**

There are many versions of Pascal available, including *Standard Pascal, UCSD Pascal,* and *Tiny Pascal.* **Pascal 80** is a TRS-80 (Models I and III only) implementation of Standard Pascal, with some restrictions and some extensions. The most important differences are the lack of pointer variables and variant records.

Efficient and compact code allows Pascal 80 to have a monitor, editor, and compiler in memory in the computer at the same time, yet leave enough room to create programs up to 23K bytes, with an additional 9K available while the program is running for variables and work space. This allows programs to be written, compiled, edited, and compiled again without time consuming disk access. This gives Pascal 80 a friendly character similar to interpreted languages such as Basic and APL, while retaining the speed of a compiled language and the unique flavor and advantages of Pascal.

Pascal 80 is ideal for modest programs, but is not intended as a development language for applications requiring memory overlays or managing very large disk files.

About this manual

This manual tells you how to use the Pascal-80 text editor, monitor, and compiler, and explains, with examples, the features of the language. While examples are given, those who have no previous experience with Pascal will probably want to review a text on Standard Pascal. Recommended texts include:

Programming in Pascal by Peter Grogono (*Addison Wesley*)

Oh! Pascal! by Doug Cooper and Michael Clancy (*Norton*)

Pascal User Manual and Report by Kathleen Jensen and Nicholas Wirth, (*Springer Verlag*)

Pascal by W. Findlay and D. A. Watt (*Computer Science Press*)

Texts which describe UCSD Pascal will be less useful. UCSD Pascal includes a number of extensions and changes to Standard Pascal. (Actually, there are many variations of even Standard Pascal.) These extensions especially involve the handling of files, character strings, and graphics. While Standard Pascal is thus a more limited language, it is an excellent introduction to computer programming in a structured language, and Pascal 80 is especially well suited to applications of moderate size.

In this manual, when you are referred to a **CONTROL** function, press the SHIFT and ↓ keys at the same time as the letter in the control function. Therefore, to get a **CONTROL Q** press the SHIFT and ↓ and Q keys at the same time.

Getting Started with Pascal 80

Pascal 80 is supplied on a single density diskette with the TDOS operating system from Micro-Systems Software. The program automatically adjusts itself for TRS-DOS 1.3 on the Model III or TRS-DOS 2.3 on the Model I, or will operate under TDOS on the Model I or DOS Plus 3.4 in the single density, double density or Model III versions.

To use Pascal on the ***Model III*** , you must have two disk drives. With **TRS-DOS 1.3** , use the *CONVERT* utility. Boot your TRS-DOS disk in drive 0, place the Pascal 80 disk in drive 1, and type (from DOS); CONVERT . After the file is converted, type PASCAL to enter the monitor. If you have **DOS Plus** on the Model III, boot DOS PLUS in drive 0, place your Pascal 80 disk in drive 1 and type *CONVERT :1* , and your Pascal 80 disk can be read directly on the Model III. Use COPY to transfer files to DOS Plus III.

To use Pascal 80 on the ***Model I*** , simply boot the disk in drive 0 and type *PASCAL*. *TDOS* has a built in lower case adapter that can be activated by pressing *SHIFT 0* . This allows you to use lower case.

To copy PASCAL 80 to another Model I DOS, you must copy each file separately. Use the DIR or CAT commands to find the names of all the files included. Then use either COPY filename:0 :1 if you have two drives or COPY1 filename if you have only one drive. Then follow the instructions on the screen.

The current version has been tested with and will work with TRS-DOS, DOS Plus, NewDOS 80 (including version 2), NewDOS, LDOS, and DoubleDOS. Since Pascal 80 has its own keyboard driver, some special features, including the screen print routines of certain systems, will not work under pascal.

To make a copy of your Pascal 80 disk on the Model I, type BACKUP, answer the questions and follow the instructions on the screen.

Known BUGs in Pascal 80

1) Array index variables in records must be global variables. Local variables do not work.
2) When you declare a variable in a record declaration, you may not reuse the same variable name elsewhere.
3) In a few places, spaces are significant where the standard specifies otherwise. For example, no space is allowed inside the assignment operator or keywords.
4) Please document any errors you find and send full information to us.
5) If anyone implements pointer variables, we would dearly love to have a copy of the new code. We are also willing to provide some assistance to those who make the attempt.

Memory MAP

START= 5300H,END= A4E0,TRA= 8409The transfer address configures for model I or III and jumps to the normal entry point at 7000H. The program text begins at A400, the stack takes 256 bytes at the top of memory and the symbol table works down from the stack.

The program is organized as follows:

5300 PCODE interpreter
EDITOR
COMPILER
PROGRAM text
SYMBOL table
STACK
(optional) INIT routines

When eXecuting a program using the X option, the pcode begins at 8000H.

Cursor Choices

Pascal 80 is supplied with a flashing underline cursor. To change this to a flashing block change memory location A0D9 (File sector 4F Byte 15) from 00 00 to 18 18. To change to a solid block change memory location A0C1 (File sector 4 E Byte FD) from C3 D0 A0 to C3 00 70.

Lower Case

Pascal 80 honors the normal Model III lower case routines. There is also a built in lower case driver for the Model I in TDOS, the disk operating system on which Pascal 80 is supplied. In either case, press SHIFT 0 to toggle between lower and upper case. Lower case letters in variable names and keywords are treated as upper case. Pascal 80 cannot distinguish between begin, BEGIN, Begin, and bEgIN. (This conforms to the Pascal standard.)

Upgrade policy

We are continually improving Pascal 80, and will be happy to provide upgrades to earlier versions at the following prices:

Upgrade Ramware Pascal 80 to New Classics Pascal 80.
Send $35 (check or money order only) and your instruction booklet.

Upgrade New Classics Pascal 80 to latest version.
Send $6 (check or money order only) and your serial number. You must have returned a registration card.

Writing a program

Once you have made a backup copy of your diskette to work with, and have put the original in a safe place, you will want to write your first program.

From DOS READY, type PASCAL and press *ENTER*. This will put you in the monitor mode and the screen should read as follows:

```
PASCAL-80
E - EDITOR
Q - QUIT (TO DOS)
K - KILL (CLEAR EDITOR)
C - COMPILE PROGRAM IN EDITOR
R - RUN PROGRAM IN EDITOR (COMPILE IF NECESSARY)
S - SAVE PROGRAM IN EDITOR
L - LOAD PROGRAM IN EDITOR
A - APPEND TEXT TO EDITOR
W - WRITE OBJECT CODE TO DISK (COMPILE IF NECESSARY)
X - EXECUTE PROGRAM FROM DISK
```

Press the E key to enter the Text Editor.

Every Pascal 80 program must have the following parts:

1) The keyword PROGRAM
   (this is usually followed by a NAME)
2) and a ; (semicolon).
3) The keyword BEGIN
4) followed at some time by the keyword END
5) and a . (period).

Now type in this program:

```
Program Print (OUTPUT);
Begin
    Writeln ('Hello, YOUR NAME')
End.
```

When you finish typing each line of a program, press the *ENTER* key. This tells the computer that the line is complete.

Substitute your own name for "YOUR NAME." Don't forget to press *ENTER* on the last line, or the editor will not remember your END. statement. If you made a mistake, just use the arrows on the keyboard to move the cursor (the flashing underscore) to the place where your mistake was and retype the line from that point.

Now we are ready to compile the program. First, press the **BREAK** key. This will put a menu at the bottom of the screen:

< QUIT TOP NEXT PREVIOUS OPEN DELETE CANCEL LINE ERASE FORMAT>

Next press *CONTROL* Q. (**SHIFT** ↓ **Q** all at the same time.) This will return you to the monitor menu:

PASCAL-80

E - EDITOR
Q - QUIT (TO DOS)
K - KILL (CLEAR EDITOR)
C - COMPILE PROGRAM IN EDITOR
R - RUN PROGRAM IN EDITOR (COMPILE IF NECESSARY)
S - SAVE PROGRAM IN EDITOR
L - LOAD PROGRAM IN EDITOR
A - APPEND TEXT TO EDITOR
W - WRITE OBJECT CODE TO DISK (COMPILE IF NECESSARY)
X - EXECUTE PROGRAM FROM DISK

You now have three different ways to compile your program. If you press ***C*** , the program will just compile. If you press ***R*** , the program will compile and execute. If you press ***W***, the program will compile and the screen will display:

ENTER FILESPEC - < BREAK > TO ABORT

If you now type in a valid filename, your program will be saved to disk.

This time, type R. The screen will display a sequence of messages similar to this:

```
00:31:32 COMPILATION STARTING
00:31:32 COMPILATION COMPLETE
00:31:32 RUNNING
Hello World
00:31:32 EXECUTION COMPLETE
PRESS< ENTER> TO GO ON
```

If you made a mistake in your program, the compilation will stop when the mistake is encountered, and the screen will display the line in which the mistake is suspected, with an arrow pointing to the possible location of the mistake, and an error message. The error might actually be in the previous line or in the command prior to the arrow. If you *PRESS< ENTER> TO GO ON* , you will return to the monitor editor. Pressing E will return you to the editor, with the line in which the error occurred being displayed. For this program, just retype the line with the error, press *ENTER*, and try again to compile it. Later on, you will want to refer to the sections on compiler error messages and editor commands.

Elements FILES and WRITELN

(OUTPUT) is a file declaration. It tells the Pascal compiler to reserve a file to output information to the screen. Actually, Pascal 80 does not require you to declare OUTPUT as a file. It has three built-in files that are always available, INPUT, OUTPUT and LP. INPUT declares that information will be coming to the program from the computer keyboard, OUTPUT declares that information will be sent to the screen, and LP declares that information will be sent to the line printer. Even though it is not necessary to declare these files in Pascal 80, it is a good habit to get into, as it is required in some other versions of Pascal, and also tells other persons looking at your program what will be happening.

WRITELN, pronounced "Write Line," writes a message to the screen. The message must be enclosed between parentheses and single quotes, as shown in our example. There is also a plain WRITE function, which in our demonstration program would make a difference that you might not even notice. The difference is that there is an extra line of space on the screen between the HELLO message and the message

00:35:30 EXECUTION COMPLETE.

WRITELN and WRITE both write messages to the screen. But WRITE will allow you to write your next message right beside your current message, while WRITELN will start the next message on the next lines.

Introduction to Editing

Let us modify our program. Go back into the editor. (Remember, press E from the monitor menu.) To see your program, you may have to type *CONTROL* T. (There is no *CONTROL* key on the TRS-80, but holding down the SHIFT and ↓ keys at the same time and pressing another key generates control characters. Thus, to get *CONTROL* T, hold down the SHIFT and ↓ keys and press the T key.) *CONTROL* T moves the cursor to the beginning of the program in the text editor. Our program currently reads:

```
Program Print (OUTPUT);
Begin
    Writeln ('Hello, NAME')
End.
```

We are going to make it read:

```
Program Print (OUTPUT);
Var count : integer;
Begin
    For count : = 1 to 50 do
    writeln (' Hello, NAME')
end.
```

LINE INSERT function

First, use the arrows to position the cursor on the B in BEGIN. Now press *CONTROL* L. This will insert a line of space in the program. Press *CONTROL* L again to insert another line of space. Now type in:

```
Var count : integer;
```

and press *ENTER*. This will add the new line to your program. (We don't need all the extra spaces in the program. They just make it easier to read.)

Now use the arrows to move the cursor on top of the W in WRITELN. Press *CONTROL* L again to insert a line here. Then type in:

```
For count : = 1 to 50 do
```

and press *ENTER*. Now press *CONTROL* Q, then R to compile and run your program. Unless you made a mistake, the computer should have printed your message 50 times.

If you had an error, try to figure out what is different between your program and the example program, particularly on the line in which the arrow is displayed on the screen or the previous line. Then press *ENTER*, then E to return to the editor to correct the error. Now press *CONTROL* Q and R again to compile your program.

The LINE ERASE function

We are going to make another change to our program by changing:

```
Writeln ('Hello, NAME')
```

to:

```
Write (count,' Hello NAME')
```

Move the cursor over top of the W in WRITELN. Press *CONTROL* E. This gets rid of our old line. Now press *CONTROL* L to insert a new line and type:

Write (count,' Hello NAME')

It would have been easier to do this by simply moving the cursor to the L in WRITELN and retyping our line, then ending it by pressing *ENTER* to put our new line in the program. However, we wanted to demonstrate the ERASE function, so we deleted our old line and inserted a new one.

Now press *CONTROL* Q, then R to compile and run the program. What was different about the output of the program?

New elements: VARiables, the assignment operator and DO LOOPs

We have added two new elements to our program. The first is a variable declaration:

```
Var count : integer
```

VAR is a Pascal keyword that tells us the information following will identify variables to be used in the program. We then declared a variable which we arbitrarily named COUNT, and told Pascal that COUNT would be an INTEGER variable. In Pascal 80, an INTEGER variable is a whole number between -32,768 and 32,767. Notice the colon (:) between COUNT and INTEGER. It must be there.

The other new element is a DO LOOP:

```
For count : = 1 to 50 do
```

The do loop tells the program to assign to the variable COUNT values from 1 to 50, counting by ones, and each time to execute the portion of the program immediately following the do loop. The := in the do loop is the Pascal Assignment Operator. It tells the computer to assign a value to a variable. In this case it is assigning the values 1, 2, 3, ... 49, 50 to the variable COUNT. You could also use it in this form:

```
COUNT : = 995
```

to assign a value of 995 to the variable COUNT.

■■■■

If you have not encountered an error yet, go back to the editor and misspell one of the keywords (PROGRAM, BEGIN, WRITELN, or END). For example, you could position the cursor over the D in END and type a space, to make the last line read EN . Now try to compile the program so you can see an error message. Don't forget to go back to the editor later and fix the error, as we will still be using the program.

Saving and Loading Programs

Let's save our program to demonstrate the disk functions of the monitor. We have two ways to save it; either as a source (text) file or as an object (compiled) file.

Return to the monitor menu. If you just ran your program, you should be able to return to the monitor by pressing *ENTER*. If you are in the editor, you will have to press *CONTROL* Q to return to the monitor.

Saving a Source File

First we will save our source program, the text that is in the editor. From the monitor menu, press S. The screen will display the message:

```
ENTER FILESPEC -< BREAK> TO ABORT
?
```

Type in a file name. A file name is any 8 or fewer letters and numbers, beginning with a letter. It may also have an extension, which is a slash (/) followed by one to three letters. It is suggested that you use **/SRC** or **/SOU** as an extension for source files and **/PAS** or **/OBJ** for compiled Pascal object files, but you may use any extension you wish or none at all. We might type:

```
HELLO/SRC
```

and press *ENTER*. The monitor will now save our source file to diskette.

Loading Source Files

Let's load our file back in. Go into the editor (E from the monitor) to make sure that nothing is there. Type *CONTROL* Q to return to the monitor. Let's load our source file. Type L and the screen will display:

```
ENTER FILESPEC -< BREAK> TO ABORT
?
```

Type in HELLO/SRC or whatever you called your file and press *ENTER*. The program should load. Check it out by going into the editor and looking. You should have your program back. (By now you should be able to do this by yourself. If not, look back through earlier examples.)

Saving Object Files

Saving a compiled program to disk is very much like saving a source program. From the monitor menu, select option W. You will again be asked:

```
ENTER FILESPEC -< BREAK> TO ABORT
?
```

This time, you may wish to use the same filename with a different extension. The computer will treat them as totally different files. Type:

```
HELLO/PAS
```

or your own file name and press *ENTER*. Your Object code will be saved to disk.

Loading Object Files

To load and run an object program from disk storage, select option X on the monitor menu. You will again be asked:

```
ENTER FILESPEC -< BREAK> TO ABORT
?
```

Type:

```
HELLO/PAS
```

or your own file name and press *ENTER*. Your Object code will be loaded into memory and run.

Although loading an object file from disk is performed almost exactly the same as loading a source file, the program functions very differently. The program is automatically executed upon loading, and once the run is over, control is returned to DOS, not to the monitor. Another major difference is that the program is loaded right on top of the compiler portion of Pascal 80. This allows the program 9K more workspace for run time variables than can be obtained with the compiler in memory.

Serial Printer Drivers

Pascal 80 will honor HIMEM at $4049 on the Model I and $4411 on the Model III if you want to add your own printer driver.

Limitations of Pascal 80

Not all of the functions of standard Pascal are implemented in Pascal 80.

The following FUNCTIONS are NOT IMPLEMENTED:

Variant Records

WITH statement

Pointer VariablesNEW, DISPOSE statements

File Window (Buffer) Variables GET and PUT procedures

Procedures PACK and UNPACK (All structures are packed on byte boundaries.)

Procedure PAGE, But try WRITE(LP,CHR(12))

Structures of files (ARRAY OF FILE, etc.)

OTHER LIMITATIONS:

Sets are limited to 256 members, and if they are numeric, must be in the range 0..255.

The identifier of a procedure or function may not be passed as a parameter to another procedure or function.

No expression passed as a value parameter may exceed 510 bytes unless it is a VAR parameter.

Integer variables used to reference array entries in a record must be global variables. Local variables will not work in this instance.

[must be indicated as (. and] as .) on the Model I. The Model III has bracket characters. To obtain the brackets on the Model III, use SHIFT ↑ for [and SHIFT @ for] . While the Model III has curly brackets , Pascal 80 uses (* and *) for comments.

Spaces are significant in a few isolated instances where they would not be in standard Pascal. Thus, the assignment operator must be : = and not : = (no space allowed).

Integer variables are limited to the range -32768 to 32767.

Extensions to Standard Pascal

Arrays of characters may be printed with a single statement. For example, if STRING is declared as an ARRAY(.1..10.) OF CHAR then WRITE(STRING) is equivalent to:

```
FOR I : = 1 TO 10 DO WRITE(STRING(.I.))
```

READ and WRITE may be used with non-text files in place of GET and PUT.

When using string constants in assignment statements and comparisons of character arrays, the constant on the right may be shorter than the item on the left and will be automatically padded with blank spaces as necessary. If:

```
VAR STRING : = ARRAY (.1..10.) OF CHAR
```

then STRING : = 'NAME' is valid and STRING > 'NA' would be TRUE. (The right hand argument must have at least 2 characters.)

PROC and FUNC are accepted as valid abbreviations for PROCEDURE and FUNCTION.

Predefined constants are MININT (-32768), PI, FALSE, TRUE, and MAXINT (32767)

REAL variables have 14 digit precision, but REAL6 (six digit precision) is provided as an optional type to save space in large arrays (4 bytes instead of 8). No time is saved, as calculations are still performed with 14 digit precision. REAL6 variables that are not members of an array or record may not be passed to a procedure or function as value parameters.

Standard files INPUT and OUTPUT need not be declared in the PROGRAM statement, and the program name is also optional. Pascal also provides another predefined file, LP, for the line printer.

Additional Procedures

The following procedures are added to standard Pascal. These are explained later under the heading "Procedures."

CLS clears the screen. CLOSE will close files. SEEK (expression,filename) will position a named file to a record within a file. INKEY returns the value of any key pressed. CALL(address,value) places a value into the A register and calls the address. It returns the contents of the A register after the call. MEM returns the number of bytes of free memory. PEEK(address) obtains the contents of a memory address. POKE(address,value) places a value into a memory location. FP(expression) returns the fractional part of a REAL number. EX(expression) returns the exponent of a REAL number to a variable. The INCLUDE function, (*$ filename *) allows you to compile a procedure from a disk source file into your program.

In addition to these built in procedures, example procedures are supplied on the program diskette for cursor positioning; GOTOXY(hpos,vpos) and for block graphics; PSET(hpos,vpos), PRESET(hpos,vpos), and POINT(hpos,vpos).

The CASE statement is extended in two ways. An ELSE clause may be included, and will be executed if no other case is satisfied. If no case is satisfied and there is no else clause, control will fall through to the next statement without an error indication.

Both REAL and INTEGER expressions are printed with the statement:

WRITE(expression:fieldwidth:digits)

A fieldwidth of -1 calls for scientific notation, and the digits parameter will be ignored, if present. A field width of 0 produces the default format, also used if no specification of format parameters is made. The default format prints the number with a space before it and as many digits after the decimal point as necessary.

EDITOR Functions in Pascal 80

Pascal 80 provides a text editor to help you prepare a series of program statements. The text which you type is collected into a text workspace in your computer's memory, which can then be compiled or saved by the monitor. Since the video display is not large enough to show the entire contents of your workspace, it functions as a movable window, displaying 15 lines of your program at a time.

The arrow keys move the cursor about the screen, When the top of the screen is reached, the display will scroll until the beginning of the program is found. In a similar fashion, when the end of the screen is reached, the display window will scroll until the end of the program is displayed.

During the entry of a program, errors can be corrected by using the *SHIFTED* left arrow key to erase characters, or by using the right or left arrows to move across the text without erasing, then retyping the line or portions of it to correct errors. The arrow keys, like all other keys, have an automatic repeat function when they are held down. Once a line is correct, save it in the text workspace by pressing the ENTER key. The entire line will be saved, even if the cursor is not at the end of the line.

You can edit a line that has already been entered by simply typing over it and pressing *ENTER*. The new line will replace the old in the text buffer. It is **VERY** important to press *ENTER*, as otherwise the changes are not saved. You can cancel the changes you have made by moving away from the line with the arrow keys before you press enter. While the screen will still display the changes, you can display the actual line by pressing *CONTROL* C (for CANCEL).

The EDITOR Command Menu

Twelve useful commands are provided in the text editor mode. While you are getting used to them, you may wish to have them listed at the bottom of the screen. To do so, simply press *BREAK*. To erase the menu, press *CLEAR*.

QUIT TOP NEXT PREV OPEN DEL CANC LINE ERA FORM BLOCK WRITE

You can use one of these commands by pressing the first letter while holding down the *SHIFT* and ↓ keys. You can use the editing commands whether or not you have this menu on the screen. It is only provided as a reference.

CONTROL:

T will position the cursor at the beginning (***Top***) of the work space.

N will display the ***Next*** 15 line page of text as if you had moved the cursor to the bottom of the page and pressed the down-arrow 15 times.

P will display the ***Previous*** 15 line page of text, if any.

Q will return you to the monitor menu. (***Quit***)

L will insert a blank ***Line*** at the current cursor position.

E will ***Erase*** the line in which the cursor resides.

F toggles the autotab function to help you ***Format*** your Pascal programs. The editor will remember the position to which you most recently tabbed, and automatically indent to that position whenever you press ENTER. This allows you to type a series of lines at the same nesting level without retabbing. Backspacing to the left of the current tab position will set a new tab position for the following lines. The autotab function is on at power up. The F command will turn it off or back on again.

C will ***Cancel*** any editing you have done on the present line if you have not yet pressed ENTER.

O will ***Open*** a space at the current cursor position for the insertion of additional text.

D will ***Delete*** a character at the current cursor position.

W will ***Write*** the program to the printer, starting at the beginning of the line that contains the cursor. It will either print to the end, or will stop if it comes to a line containing only the @ command for a text marker. Thus you can print anything from a full program to a small portion.

B is the ***Block*** move command, which can be used in conjunction with the Append command from the monitor. Append adds another file to the end of the current file. Control B will move all text from the Block marker to the end of the file to the position indicated. First place a text marker, the @ sign, on a line by itself right before the block to be moved. Then move the cursor to a blank line where you wish the new text to be inserted and press Control B. The block of text will be moved, and the marker line with the A symbol will be erased at the same time.

To move only a portion of the text, you must make two moves, as this command moves everything from the marker to the end of the text. After you make your first move, simply mark the end of the new section and use another block move to place the rest of your material in its proper location.

The Pascal 80 Monitor

The first display given to you upon starting up Pascal 80 is a menu of commands, as follows:

```
                PASCAL-80
E - EDITOR
Q - QUIT (TO DOS)
K - KILL (CLEAR EDITOR)
C - COMPILE PROGRAM IN EDITOR
R - RUN PROGRAM IN EDITOR
S - SAVE PROGRAM IN EDITOR
L - LOAD PROGRAM IN EDITOR
A - APPEND TEXT TO EDITOR
W - WRITE OBJECT CODE TO DISK
X - EXECUTE PROGRAM FROM DISK
```

These commands constitute the Monitor, a " *master control* " program for Pascal 80. In each case, all you have to do is press the appropriate key to start the operation. *Command* ***K*** will verify that you really mean to destroy the program in the editor by asking:

ERASE TEXT? (Y/N)

You must reply by pressing " ***Y*** " to actually erase the text. For commands ***S, L, A, W,*** and ***X*** , the screen will display the question:

ENTER FILESPEC - <BREAK> TO ABORT
?

You must answer by typing the name of the file you want to load, save, or add to your current program. A file name is of the form *FILENAME/EXT.PASSWORD:d* where:

FILENAME is from 1 to 8 letters and numbers you choose to identify your file. The first character must be a letter from A to Z. Remaining characters may be either letters or numbers from 1 to 9. No spaces, punctuation marks or other symbols are allowed.

/EXT is an optional extension that you may use to identify your program. You may choose to omit the extension entirely. An extension consists of a slash (/) followed by 1 to 3 letters and numbers. The first character after the slash must be a letter.

.PASSWORD is an optional period, followed by 1 to 8 letters and numbers you may want to use to prevent unauthorized access to your files. As with filenames, the password must begin with a letter, but both letters and numbers are legal in the rest of the password. When a user asks the disk operating system to give a directory of the disk, only the filename and extensions are shown, although password protected files are indicated by a " *P* " after the file name. The user must know the password to gain access to the file. Do not place much confidence in passwords. There are many ways to defeat them, and they generally only stop computer novices from reading the file.

:d is the drive number of the disk drive that holds your file. For the majority of users, that will be either *:0* or *:1* , though more than two drives may be present. Some hard disk systems for the TRS-80 may appear to the system as 90 or more disk drives, so *:90* is possible.

Monitor Functions

E places you in the editor mode. If there is text already in the editor, you will be able to see it. If the computer has just stopped in the middle of compiling a program because there is an error in your program, when you go back to the editor you will be on the line in which the computer discovered the error, making it easy for you to correct the error. Note that in some cases, including times when the previous line is missing a semicolon (;), the error will actually be on a previous line. A preceding section of this manual discusses the editor commands.

Q returns you to the disk operating system. This allows you to execute disk commands like *DIR* (get a directory of files on the disk), *FREE* (find out how much space is left on the disk for new files) or *PRINT* (print a file on the line printer). You can use **Q** to go to DOS and still return to Pascal without disturbing your program. If you want to restart Pascal, just type *PASCAL* , and any program in memory will be cleared. If you want your program back, type *PASCAL* and hold down the *ENTER* key until Pascal 80 loads. The program in the editor will not be disturbed. You may then type **E** to see or modify your program.

K erases the contents of the editor. Since doing this by accident could result in the loss of a lot of programming time, this option also includes a chance to change your mind by asking if you want to *ERASE TEXT? (Y/N)* . If you type " **Y** " the program in the editor will be cleared. Any other key pressed will return you to the monitor mode without erasing your program.

R will run the program that is currently in the editor, known as the source program. If the program has not yet been compiled, or you have returned to the editor since the last time you ran the program, it will automatically be compiled before it is run.

S will save the source program that is currently in the editor to a disk file. You will be asked to provide a file specifier. (You may find it useful to add the extension */SRC* or */SOU* to your filename, so that you can later distinguish source files from other types of files.) If you do not want to save the current file, press *BREAK* instead of giving a file name, and you will be returned to the main menu.

L will load a source file from disk to the editor. You will be asked for the name of the file you wish. If you do not remember the filename, use the **Q** option to go to the disk operating system, and type *DIR* to obtain a directory of the files on the diskette. You can then type *PASCAL* and press *ENTER* to return to the monitor and then load your program. If you do not want to load a file, you can press *BREAK* to return to the monitor menu.

A will load a source file from disk into the editor without disturbing a file that may already be in the editor. The new file will be added on to the end of the current file. You should be aware that if the editor contains a complete program, the compiler will stop compiling as soon as it comes to the first " *END.* " statement it finds, so check your source code when appending a file. You will be asked for the file name of the file you wish to append to the current source program. You may return to the monitor without disk access by pressing BREAK instead of giving a filename. If you attempt to load a file that is not a Pascal source file, you will receive the error message *BAD FORMAT.*

If you wish to add a file from disk in the middle of a current file, you can either use the ***INCLUDE*** procedure or the block move command in the editor.

W will save an object file to disk. If the current source program has not been compiled, it will be compiled and then you will be asked to give a file name. You may abort the procedure by pressing *BREAK*. *You may wish to add the extension /OBJ* or */PAS* to your file name so that you can later recognize the file as a complied program.

Saving a program in compiled form will allow you to run the program in memory that would otherwise be occupied by the editor and compiler, giving you 9,000 bytes of additional memory during execution that can be used for variables and workspace. A compiled program also takes less room on the diskette for storage, and saves the time it takes to compile it each time you run it. This may also make it possible for you to create programs that will run in a 32K computer, even though they take a 48K computer to compile.

X will allow you to directly execute a compiled object program from disk. You will be asked to supply a file name, and may abort the procedure by pressing *BREAK*. The object file will be loaded right over portions of the editor and compiler of Pascal 80, saving memory space but requiring you to reload Pascal 80 if you wish to use it again. For this reason, after your program is executed, you will be returned to DOS, not to the monitor. If you attempt to load a file that is not a Pascal object file, you will receive the error message *BAD FORMAT.*

Compiler Options

Six optional compiler instructions will allow you to route compiler output to the printer, suppress compiler output (except error messages), indicate the amount of stack space and symbol table space available to the compiler on each line, print each byte of compiled object code as it compiles, zero all variables before execution, and/or verify all disk file operations.

The option instructions are **HARDCOPY, NOLIST, MEMORY, CODE, ZERO** and **VERIFY**. They may be abbreviated as **H, N, M, C, Z** and **V**, as the compiler only checks the first letter. However, they must be in upper case. Simply insert the appropriate instruction or instructions at the very beginning of your program. The compiler considers anything found before the keyword PROGRAM to be a compiler instruction. Instructions can be separated by any valid delimiter, including space, carriage return and slash, so that *H/C/Z* is the same as *HARDCOPY CODE ZERO.*

The first four options affect the listing of the program, as follows:

HARDCOPY or **H** sends all listings to the line printer.

NOLIST or **N** suppresses the compiler listing, except that compile time error messages are still printed.

MEMORY or **M** adds two more hexadecimal numbers to each line of compiler output to indicate compiler stack memory and symbol table memory available. The output will appear in the following form:

0000 00F4 5C4B PROGRAM; BEGIN END.

The first number (0000) tells that 0 bytes have been compiled before this line. The second number (00F4) tells that F4 (244 decimal) bytes of work space are available on the compiler stack. The third number (5C4B) reveals that there are 5C4B (23627 decimal) bytes of symbol table space available.

CODE or **C** tells the compiler to print out each byte of compiled code as hex numbers in the compiler listing. This is not a complete listing. In some situations, such as forward jumps, the compiler will generate dummy place holders to be filled in later with the correct values. In addition, the pseudocode is self relocating, and relative addresses generated by the compiler will be replaced by absolute RAM addresses prior to the execution of each block.

The remaining two compiler options affect the execution of a program:

VERIFY or **V** will verify all write operations to disk files. Pascal 80 overrides any verify instruction given under the DOS VERIFY command.

ZERO or **Z** causes all variables in a program compiled with this option to be set to zero before execution. If you do not use this option (or explicitly set your variables in the program), then your variables will contain what ever is left in memory from your previous program. This can lead to strange results, including the printing of garbage instead of your data.

Pascal keyword *FORWARD* is a compiler option. *FORWARD* is used to declare a procedure or function in advance of its actual appearance, so that references to the procedure or function will not generate a compile time error message. Any parameters must be declared at the time of the *FORWARD* reference. *FORWARD* must be preceded and followed by a semicolon. Here are some examples:

```
Procedure Anything; FORWARD;
Procedure Interest (Amount,Rate,Days : real); FORWARD;
Function Even (var Number : integer): boolean; FORWARD;
```

INCLUDE (*$ filename *)

The *include* procedure allows you to either compile a whole program from disk or to compile procedures into a program in the editor.

For example, if you wrote a program so large that there is not enough room to compile it, you could save it to disk as a source file and compile it with the include procedure. If it was named PROGRAM/SRC, you would clear the text editor (with the K command from the monitor menu) and place this in the editor:

```
(*$ PROGRAM/SRC *)
```

Nothing else need be in the editor, though you could include compiler options. When you return to the monitor and select the C, R, or W options, the program will be compiled from disk instead of memory.

If you wish to include procedures in your programs, save the procedure as a source program. See the demo program COINTOSS/SRC for examples. Here is a simple use:

```
program IncludeProcedure (Output);
(*$ GOTOXY/SRC *)
begin
 cls;
 gotoXY(5,6);
 write('Here it is.');
end.
```

You may wish to have the actual source code in your program, instead of using include when you create an object file. If so, append your procedure from disk to the end of your program using the A option from the monitor menu. Then enter the editor and use the block move command to place the procedure where required.

Pascal 80 Constants

Pascal 80 has 5 built in constants, ***True, False, MinInt*** (the smallest allowable integer), ***MaxInt*** (the largest allowable integer), and ***Pi***. In addition, you can declare your own constants. The following sample program and sample run will demonstrate defining a constant and show each of the built in constants.

```
Program Constants (Output);
Const  Two = 2.0;
Var Diameter,Circumference : real;
Begin
 Writeln(Two);
 Writeln(True);
 Writeln(False);
 Writeln(MinInt);
 Writeln(MaxInt);
 Writeln(Pi);
 Diameter:= Two;
 Circumference:= Pi * Diameter;
 Writeln( Circumference );
End.
```

Sample Run:

```
 2
TRUE
FALSE
-32768
 32767
 3.1415296535898
 6.2831853071796
```

Variable Types

Pascal 80 allows 6 types of variables, ***Boolean, Integer, Char, Real, Real6,*** and ***Text***. In addition, you may define your own types with the *Type* statement. Global variables must be declared at the beginning of your program, and can be used throughout the program. Local variables may be declared within a function or procedure, and will only work within that function. Thus, you can even use the same variable name in different procedures without conflicts. Variable names may be any length, and all characters in the name are significant; *VariableNumberSeventyFour* and *VariableNumberSeventyFive* will be recognized as different variables.

Boolean variables are either *True* or *False*.

Integer variables must be whole numbers between -32,768 (*MinInt*) and 32,767 (*MaxInt*).

Char or character variables are letters, numbers, symbols, spaces, and punctuation marks. To get lower case on the Model III, you must press *SHIFT 0*. If you have lower case in your Model I, TDOS has a built in lower case driver, and you can either *SHIFT* for lower case or use *SHIFT 0* as in the Model III. You can also generate characters using the ***CHR*** function.

Real variables have 14 digit precision, and all calculations, including logarithmic, trigonometric and arithmetic functions are performed with 14 digit precision.

Real6 variables, with six digit precision, may be used to save space in large arrays, using four bytes per variable instead of eight. However, no time is saved, as 14 digit precision is still used in all calculations. In addition, there are limits to the use of REAL6 variables. Unless they are members of an array or record, REAL6 variables may not be passed to a procedure or function as value parameters.

A ***Text*** variable is a packed file of characters.

Sample Variable Declarations:

```
program Variables (Input, Output);

  Const  Length = 16;
  Type   Number = Array [0..Length] of Integer;
         SetC = Set of char;
Var    N : Number;
       C : Char;
  I, J, K : Integer;
       B : Boolean;
       S : SetC;
       R : Real;
     R6 : Real6;
    Flag : Boolean;
       A : Array[0..3] of number;
       T : Array[1..1000] of char;...
```


Functions and Operators

It can be easy to confuse functions and procedures, especially those that are built into the language. While both are essentially subroutines, the distinguishing characteristic of a function is that it returns a single value. Pascal includes arithmetic and logic operators, including trigonometric and logarithmic functions, conversion, file handling and ordinal functions, as well as a few other functions. An operator is similar to a function, and also returns a single value, but usually is binary, requiring an operand on either side, as in 2 + 2.

Arithmetic Functions

Arithmetic Operators (+ - / * DIV MOD)

The arithmetic operators include + for addition, - for subtraction, * for multiplication, / for real division, ***DIV*** for integer division, and ***MOD*** to obtain the remainder of an integer division.

```
program Arithmetic (Output);
var Two, Three : Real;
  Four, Five : Integer;
begin
  Two := 2.0; Three := 3.0
  Four := 4; Five := 5;
  write(Two × Three , Four + Five);
  write(Two - Three , Four - Five);
  write(Two * Three , Four * Five);
  write(Two / Three);
  write(Four Div Five , Four Mod Five)
end.
```

ABS, SQR, and SQRT

The ***ABS*** function gives the absolute value, ***SQR*** gives the square, and ***SQRT*** gives the square root of a number. These functions will work on Real, Real6, and Integer variables.

```
program Numbers (Output)
var    R : Real;
      R6 : Real6;
       I : Integer;
begin
  R := -1.414; R6 := 81;
  For I := -3 To 3 Do Writeln(ABS(I));
  writeln(Sqr(R));
  writeln(SqRt(R6));
end.
```

Comparison Operators (=, <>, <, <=, > =, >)

The comparison operators are equal = , greater or less than <> , less than < , less than or equal to <= , greater than or equal to >= , and greater than > . They may be used in comparisons of all types of variables.

```
program Compare (Output);
var One,Two : Real;
   String : Array[1..6] of Char;
begin
   One := 1.0; Two := 2.0; String := 'String';
   write(One = One , One = Two);
   write(One <> One , One <> Two);
   write(One < Two , Two < One);
   write(One <= Two , Two <= One);
   write(One >= Two , Two >= One);
   write(One > Two , Two > One);
   write(String = 'String');
   if String > 'St' then
      write('The whole is greater than the part)
end.
```

The Assignment Operator (:=)

In Pascal, the equal sign is a comparison operator, although it is also used to identify types and assign values to constants. To assign values to a variable, the assignment operator, a colon followed by an equal sign, is used. According to the ISO Pascal Standard, spacing should not matter here. However, Pascal 80 does not allow a space between the colon and the equal sign.

```
VariableName := Value
```

Logic Operators (AND OR NOT)

Pascal allows the standard boolean logic operators **AND, OR** and **NOT** . Logic operations should use parentheses not only for syntax, but also to make the logic clear to human readers.

```
program Logic (Output);
begin
   writeln( (True AND False) );
   writeln( (True OR False) );
   writeln( NOT(True) );
   writeln( Not(True AND False) )
end.
```


Logarithm Functions

EXP and LN

The logarithm functions are **EXP(x)** to raise the natural logarithm e to the base x, and **LN(x)** to represent the natural log of an integer or real number x. While Pascal does not have an exponentiation operator, these functions can be used in combination to do so, as shown in the example.

```
program Powers (Input, Output);
var Power : integer;
    Number, Result : real;
begin
   readln(Number);
   readln(Power);
   Result := exp(Power * ln(Number));
   writeln(Result);
end.
```

Trigonometric Functions

SIN and COS

The trigonometric functions of Pascal 80 use radians instead of degrees. This program illustrates converting from degrees to radians, as well as the use of SIN and COS:

```
program TrigFunctions (Input, Output);
var Degrees,Minutes,Seconds : integer;
   Radians : real;
Begin
   writeln('Designate an angle');
   write('Degrees ');
   readln(Degrees);
   write('Minutes ');
   readln(Minutes);
   write('Seconds ');
   readln(Seconds);
   Radians := Pi * (Degrees + Minutes/60
      + Seconds/3600) / 180;
  writeln('The sine is,'sin(Radians):10:5);
  writeln('The cosine is,'cos(Radians):10:5);
end.
```

ARCTAN

ArcTan functions in a similar fashion to the Sin and Cos functions:

```
program ArcTanDemo (Input, Output);
var Tangent,Degrees : real;
begin
   write('What is the Tangent');
   readln(Tangent);
   Degrees := ArcTan(Tangent) * 57.29578;
   write('The angle is ',Degrees:6:1,'degrees.');
end.
```

ArcSin, ArcCos, and Tan

Standard Pascal does not have dedicated functions for the ArcCosine, ArcSine, and Tangent, but they may be derived from the existing functions as follows:

```
function Tan(x:real):real;
begin
  Tan := Sin(x)/Cos(x)
end;

function ArcSin(x:real):real;
var temp : real;
begin
 temp := sqrt(1-x*x);
 if temp = 0 then
    begin
       if x<0 then ArcSin := -Pi/2
       else ArcSin := Pi/2
    end
 else ArcSin := ArcTan(x/temp)
end;

function ArcCos(x:real):real;
begin
 if x = 0 then ArcCos := Pi/2
 else if x>0 then ArcCos := ArcTan(sqrt(1-x*x)/x)
    else ArcCos := ArcTan(sqrt(1-x*x)/x) + Pi
end;
```


Memory Management Functions

PEEK(address)

Peek returns the contents of the address, type *integer*.

```
program Example2 (Output);
begin
 if (peek(12342) = 195)
 then write('This is a Model III')
 else write('This is a Model I')
end.
```

CALL(address,value)

Call places a value between 0 and 255 into the A register and calls the address. It returns the contents of the A register, type *integer* , after the call. You must assign a variable to receive the contents of the A register, as in the example, whether or not you wish to use it.

```
program Example3;
var Address,Byte,Name : integer;
begin;
 Address := 73; Byte := 0
 write('PRESS ANY KEY');
 Name := CALL(Address,Byte);
 writeln(Chr(Nome))
end.
```

MEM

Mem returns the number of bytes of free memory, type *integer*.

```
program Example4 (Output);
var Memory : integer;
begin
 Memory := MEM;
 write(Memory)
end.
```

File Handling Functions

INKEY

InKey returns the value (type char) of any key pressed. If no key is pressed, chr(0) is returned.

```
program Example1 (Input, Output);
var Letter := Char;
begin
 repeat Letter := inkey until ord(Letter)> 0;
 writeln(Letter)
end.
```

EOF and EOLN

Functions **eof** (End of File) and **eoln** (End of Line) are discussed in conjunction with the procedures *read* and *readln* , and in the File section of this manual.

Ordinal Functions.

ORD, PRED, and SUCC

Integer, boolean and ***char*** variables share a common characteristic not found in *real* ***values. Each of them has a defined set of possible values, and each has a fixed order and may have a predecessor and successor in that set of possible values.*** **Ord** returns the position in the data set, **pred** returns the variable preceding the current position in the data set, and **succ** returns the next variable in the data set. Pred(MinInt) and succ(MaxInt) fall outside the range of integer variables and stop the program with error messages.

```
program Ordinals (Output);
var I : Integer;
    C : Char;
begin
 I := -32000; C := 'Y';
 writeln(pred(I),succ(I),ord(I));
 writeln(pred(C),' ',succ(C),' ',ord(C));
 writeln(pred(True),succ(True),ord(True));
 writeln(pred(False),succ(False),ord(False))
end.
```

CHR

Chr will print a character from its ordinal number in the ASCII character set.

```
program Chr;
var ASCII : integer;
     BigChr : char
begin
 BigChr := CHR(23);
 cls;
 write(BigChr);
 for ASCII := 32 to 191 do
     write(chr(ASCII),' ')
end.
```

Conversion Functions

ODD, ROUND, and TRUNC

Odd is a Boolean function that is *True* when its argument is an odd integer. **Round** will round a *real* number to the nearest integer value. **Trunc** will truncate a *real* number by dropping the fractional part.

```
program Numbers (Output);
var   N : real
     I : integer
begin
  R := 5.6;
  writeln(round(N),trunc(N));
  for I := 1 to 10 do
  begin
   if odd(I) then writeln(I,' is odd.)
  end
end.
```


Set Membership, Union, Difference and Intersection
IN, +, - and *

In is used to check set membership. The plus sign is used to join two sets (*union*), the minus sign to isolate the elements of one set not common to the other set (*difference*) and the asterisk to find the common elements (*intersection*) of two sets.

```
program InDemo; (Output)
type S= set of 1..100;
var A,B,Union,Intersection,Difference : S;
    I : integer;
begin
 A := [3..5];
 B := [5..10];
 Union := A + B;
 Intersection := A * B;
 Difference := A - B;
 write('Union set='); for I := 1 to 100 do
   if I in Union then write(I);
 writeln;
 write('Intersection set='); for I := 1 to 100 do
   if I in Intersection then write(I);
 writeln;
 write('Difference set='); for I := 1 to 100 do
   if I in Difference then write(I)
end.
```

Set Comparison Operators

The ordinary relational operators, except for < and>, can be used in set operations. The operators are:

```
=   set equality; two sets identical
<>  set inequality; sets are not identical
>=  set contains another set
<=  set is contained by another set
```

```
program SetCompare;
type JunkFood= set of (Franks,Burgers,Fries,Sodas,Pizza);
var BurgerPrince : JunkFood;
   PizzaPrince : JunkFood;
   MacFrank : JunkFood;
begin   BurgerPrince := [Burgers..Sodas];
   PizzaPrince := [Sodas..Pizza];
   MacFrank := [Franks..Sodas];
   writeln;
   writeln('   Pizza Prince and Burger Prince');
   writeln('Equality - ',(PizzaPrince= BurgerPrince));
   writeln('Inequality - ',(PizzaPrince<> BurgerPrince));
   writeln;   writeln('   MacFrank and Burger Prince');
   writeln('Includes - ',(MacFrank>= BurgerPrince));
   writeln('Is Included - ',(MacFrank<= BurgerPrince));
end.
```

WRITE and WRITELN

Write and **WriteLn** print material to a file, including disk files and the built in files *Output* (the screen display) and *LP* (the line printer). Output is the default device, so that Write('This') will print the word This on the screen. Writeln terminates the printing with an end of line character (0D hex or 13 decimal). This serves as an *eoln* terminator for a disk file or causes the printer or display to move to the next line. If you use Write, subsequent Write statements will be appended next to the first one, with no characters or spaces in between.

To write to a named file, use the Pascal file name as the first argument in the write statement. Write(LP,'Test') will print the word Test on the line printer. The use of Write and Writeln in text and record oriented files is described in the section on files.

Both *real* and *integer* expressions are printed with the statement *write(expression:fieldwidth:digits)*. A fieldwidth of -1 calls for scientific notation, and the digits parameter will be ignored if present. A field width of 0 produces the default format, also used if no specification of parameters is made. The default format prints the number with a space before it and as many digits after the decimal point as necessary, up to the maximum precision of the computer, 14 significant figures. Field width and digits parameters are interpreted modulo 256.

Example of formatted write statements:

```
program Example (Output);
var Number : real;
begin
 Number := 12345.98765
 writeln(Number);
 writeln(Number:-1);
 writeln(Number:5:0);
 writeln(Number:10:0);
 writeln(Number:10:2);
end.
```

Printed output can be frozen by holding the *CLEAR* key during execution, or the space bar during compilation.

CLS

CLS clears the screen:

```
program Example6 (Output);
begin
 cls
end;
```


READ and READLN

Pascal was originally designed to receive input from punch cards, not a terminal keyboard. Some modifications have been made to the functions *read, readln, eoln,* and *eof*, so that they work differently when receiving material from the keyboard instead of a disk file. If no file name is specified, these functions will read from the keyboard: *read(variable)* is the same as *read(input,variable)*. The only difference between read and readln on keyboard input is that readln moves down to the next line of the display after receiving input while read continues at the next character location. This means that if you use read, you can type in several inputs, separated by a space, on the same line. In disk files, readln skips to the next EOLN marker, so that any intervening data is passed over.

Reading text from the keyboard is somewhat clumsy due to the need to individually place characters in a variable:

```
program Read1 (Input, Output);
var   C : Char;
begin
 repeat
  read(C);
  write(C)
 until eoln
end.
```

In this program, you could not use:

```
 while not eoln do read(C)
```

because the starting condition in a Pascal 80 *read* is an eoln character. (See the sample program "COINTOSS./SRC for sample read procedures.)

```
program Read2 (Input, Output);
var   C : array[1..16] of char;
      I : integer;
procedure Input;
begin
  I := I + 1;
  read(C[I]);
end;
begin
  I := 0;
  C := ' ';
  repeat Input until EOLN;
  writeln(C)
end.
```

In Pascal 80, **eof** is true if and only if the next character to be read is a special eof marker produced by pressing the *CLEAR* key. Eof is printed as a graphics square (hex 8F or decimal 143) and should appear at the end of a line, after all data. For teaching purposes, this can be used to simulate a punched card deck and allow you to use programs written for stream oriented input without conversion.
Likely error messages with READ statements:
READ PAST EOLN - You attempted to read a character after the special **eoln** character.
REDO - You attempted to read an illegal character into a numeric variable (+,-,digits) or a number outside the range (-32768 to 32767) into an integer variable. You can simply reenter the correct data.
You can interrupt a program during an input statement by pressing the BREAK key. This will produce the message ***BREAK AT 0000*** .

CLOSE

Close without parameters will close all open files. **CLOSE(filename)** will close a specific file. All files are automatically closed when a program stops execution or if an error occurs.

```
program Example8 (Fila :' FILE/DAT:0');
var Fila : text;
    Message : Array[1..26] of char;
begin
 Message := 'Example of Closing a File';
 write(FILA,Message);
 close(FILA)
end.
```

SEEK(expression,filename)

Seek will position the named file to the record whose number is given by the expression. Records are numbered starting with 0. If necessary, the file will be reopened before the seek. Seek may only reference the first 65535 bytes of a file.

```
Seek(Record,FILEB);
```

POKE(address,value)

Poke places a value between 0 and 255 into a memory location. Use decimal addresses, and subtract 65536 from addresses greater than 32767, as in the Basic POKE instruction. Thus 8000H is -32768 and 9000H is -31746, etc:

```
program Example7;
begin
 poke(15365,65)
end;
```

Poke to an address below 512 is reserved for graphic functions, explained later under graphics.

CASE and ELSE

The **Case** statement is extended in two ways. An **ELSE** clause may be included, and will be executed if no other case is satisfied. If no case is satisfied and there is no else clause, control will fall through to the next statement without an error indication.

```
program DemonstrateCase (Input, Output);
var ch : char;
  Stop : boolean;
procedure GetChar;
begin
  write('How would you answer a yes or no question? ');
  read(ch);
  writeln;
  case ch of
    'Y,'y' : Writeln('You answered yes.');
    'S,'s' : Begin
               Stop := True;
             Writeln('You want to stop.')
             end;
    'N,'n' : Writeln('You answered no.')
    else writeln('I don't understand you!')
 end
end;
Begin
  Stop : = False;   repeat GetChar until Stop
end.
```

EX(expression) and FP(expression)

EX returns the exponent of a *real* number to a variable of *type integer* , while **FP** returns the number as a fraction to that exponent, *type real.*

```
program Example5 (Output);
var Number : real;
begin
 Number := 98.123;
 write(EX(Number));
 write(FP(Number))
end.
```

Files

Pascal 80 offers two kinds of files, **Text** files and record oriented (**File Of** ...) files. If your program uses disk files for input or output, Pascal requires you to declare these files in the Program statement. Because the rules for Pascal identifiers are different from the rules for TRS-DOS file specifiers, Pascal 80 allows you to equate a Pascal file identifier with a TRS-80 file specifier, using the following notation:

```
program Example (FileA : 'DATAFILE/DAT:1');
```

IF you choose to use this format, any reference to FileA in your program will actually identify DATAFILE/DAT on drive 1. If you wish to stick to Pascal format, you may use this type of file declaration:

```
program Easier (FILEA,FILEB);
```

This will give you FILEA and FILEB as the actual TRS-DOS filenames.

Pascal requires you to also declare your file names as variables, with the identifier TEXT for text files and FILE OF ... for record oriented files:

```
program TextFile (FILA);
var   Fila : Text
```

or

```
program RecordFile (FileB);
var     FileB : file of real
```

To write an expression to a text file, use the format *Write(filename, expression)*. In Pascal 80, it is not necessary to explicitly open the file. *Write* will automatically open the file, and will even create it if it does not exist on the disk. A write statement always adds text at the end of a file. It will advance to the end of the file before writing. You can use fieldwidth and digit parameters in writing to a file in the same way that you use them on the screen. (See Write in the *Function* section.) It is also permissable to write several expressions in the same Write statement:

```
Write(filename,expression1,expression2);
```

Writeln(filename,expression) will work the same as Write, except that an end of line marker (EOLN, ASCII 13, or Carriage Return) will be written at the end of the expression.

Read(filename, variable) reads a number from the file into a numeric variable, or a single character into a character variable. If the file exists but is not open, Pascal 80 will open it and starts reading at the beginning. Subsequent read statements will continue where the previous read statement left off.

Two error messages are likely when reading files. A *FILE NOT FOUND* error means that you tried to read a file that is not on that disk. A *MISMATCH* error occurs if you mix types, and try to read letters into a numeric variable or a real number into an integer variable.

Readln(filename,expression) will read a number or one character from a file and then skip to the next end of line marker.

Reset(filename) closes a file and reopens it at the beginning. The next read statement would read the first character or number, but a write statement would skip to the end of the file. If a file does not exist on a disk, a RESET statement will create the file.

Close(filename) will close an individual file. ***Close*** without a filename will close all open files. Files will also close automatically if an error occurs or the program stops execution.

ReWrite(filename) will kill a file and release the space on the diskette, then open a new, empty file with the same name.

EOF(filename) is *True* if the file is positioned at an end-of-file marker, 8F hexadecimal. If the file is closed, *EOF* will open it, and return a *False* unless it is an empty file.

EOLN(filename) is *True* if the file is positioned at an end of line marker, 0D hexadecimal. If the file is closed, it will be opened.

Pascal 80 also lets you use read and write with non-text files. The syntax is:

```
Write(filename,variable,variable...);
Read(filename,variable,variable...);
```

Both the file name and the variables must be of the same type. If we have the following declarations:

```
type BigOne = array[1..50] of real;
var  FileA : file of BigOne;
     VarName : BigOne;
```

NOTE: Model I users will have to substitute (. for [and .) for]

This will set up the file to allow us to write the variable to the file with:

```
write(FileA,VarName);
```

The Write statement will open or create the file if necessary. Subsequent Write statements will add to the end of the file.

Read(filename,variable) will read the variable from the file. The file will be opened if necessary, but will not be created if it does not exist. The file pointer is advanced after each read.

Seek(expression,filename) will position the file to the record whose number is given by the expression. The first record in each file is numbered 0 and the second record is record 1. If necessary, the file will be opened before the seek. It is not possible to seek beyond byte 65,535 in a file.

Reset(filename) may be used with record-oriented files and is equivalent to Seek(0,filename).

ReWrite(filename) and ***Close(filename)*** work in the same fashion on record-oriented files as they do on text files.

EOLN, EOF, Writeln, and Readln are undefined in record oriented files. Attempting to read beyond the end of a file will give undefined results.

Notice the difference between record-oriented files and text files in the functioning of the *write* function. In a text file, data is always added at the end, but in a non-text file reading and writing occur at the same place, and the file pointer must be set by a *Seek* or *Reset* statement. This allows you to update a file by overwriting the individual variable.

It is possible to close a file and reopen it as a different type. For example, you might want to read a *text* file as a *file of char*.

Your Pascal 80 diskette contains programs with the filenames MAILIST/SRC and CREATE/SRC to demonstrate text file input and output. You may wish to load them into the editor and examine them to see how they work.

Utilities to Pack and Unpack Files

All Pascal 80 source files are stored in a compressed manner. Wherever blank space appears on the screen in a program listing, the disk file will store a single byte of code to indicate the number of spaces. (The code is 80 Hex plus the number of spaces.) While this saves space on disk and in memory, it does make it difficult to use Pascal 80 with another editor, such as Radio Shack's excellent Scripsit, or to transfer Pascal 80 programs to other computers and other Pascal compilers.

To overcome this problem, two machine language utilities are provided on your Pascal 80 disk. **ASCII/CMD** will convert a Pascal 80 file to an ASCII file. **TEXT/CMD** will convert an ASCII file to a Pascal 80 file. The syntax for either is quite simple. You must first save the source file to disk. Exit Pascal 80 using the Q option from the monitor menu and type:

ASCII filename1 TO filename2 (or)
TEXT filename1 TO filename2
Thus, the command:

ASCII YOURPROG/SRC:0 TO YOURPROG/ASC:1
would convert the Pascal 80 file YOURPROG/SRC on drive 0 to an ASCII file on drive 1.

ASCII files have been tested with Scripsit. There are some limitations. ASCII and TEXT do not provide an end of file character, so if your editor looks for a HEX 00 or other special character at the end of a file, you will have to append one.

GRAPHICS

There are three different ways to produce graphics in Pascal 80. You can use the POKE procedure to directly place graphics characters in screen memory, which runs from addresses 15360 to 16383 on the TRS-80. You can use the CHR function to write graphics characters to the screen. The disk contains procedures that you can INCLUDE in your programs to give you the equivalent of Radio Shack Basic's SET, RESET, and POINT graphics. The program COINTOSS/SRC on your disk will demonstrate these programs, which are explained later.

```
Program GrafDemo;
Begin
  CLS;
  Write(' ', CHR(141));
  Poke(15900,191);
End.
```

Graphics and Random number extensions

PSET, PRESET, POINT, RND(N), and RNDR.

These functions are equivalent to the Radio Shack Basic commands SET, RESET, POINT, RND(N), and RND(0). Sample procedures are on the disk to show you how to use these functions. The graphics functions are accessed with the procedure GRAPHIC/SRC and the random number functions with the procedure RANDOM/SRC. The COINTOSS/SRC program on your disk uses the INCLUDE procedure to incorporate both of these procedures and the GOTOXY procedure in a program.

RND(N)

This procedure will return a pseudo-random integer result between 1 and N. The procedure RANDOM/SRC on the disk shows how to use both **Rnd(N)** and RndR.

RNDR

While Rnd(N) produces an *integer* result, **RndR** will produce a *real* number between 0 and 1. RNDR does not take an argument. The new random seed is produced as follows:

*(477 * Old seed × 3461) MOD 32768*

Procedures **pSet**, **pReset** and **Point** are implemented as sub functions of the **Poke** procedure. Since it is not possible to change ROM with the poke command, pokes to locations below 512 have been reserved for graphics. From Poke(0,0) to Poke(127,47) functions as Set with the same argument. For Reset, use Poke(horizontal position + 128, vertical position). For point, use Poke(horizontal position + 256, vertical position), then Peek(21458) for the result of zero if the point is off and non-zero for on. These functions are demonstrated on the file GRAPHIC/SRC.

Creating Command Files

The author package on your Pascal disk will allow you to create files that will execute directly from DOS. These files will appear to the end user as if they were machine language files, and you do not have to pay any royalty if you sell these programs. (Note: See the conditions under licensing in the front of the manual.)

The author package consists of two parts, a machine language program called ***AUTHCODE/CMD*** and a Pascal 80 source file titled ***AUTHOR/SRC***. To use the author package follow these steps:

1) Create your program using Pascal 80.
2) Compile your program and save it to disk using the **W** option from the monitor menu.
3) Return to DOS, type **AUTHCODE**, and press *ENTER*.
4) Return to Pascal 80 by typing **PASCAL** and pressing *ENTER*.
5) Using the **L** option, Load **AUTHOR/SRC**.
6) Press **R** to compile and run AUTHOR/SRC.
7) Follow the instructions on the screen. You will be asked:
   a) If you need to load AUTHCODE/CMD
   b) If you have compiled your file
   c) What name you wish to use for your new file
8) If you answer **N** then **Y** and give a filename that is not currently on the disk, you will then be asked if you are ready to continue. Answer **Y**.
9) Now you will receive further instructions. Read them and press *ENTER*.
10) This will return you to the monitor menu. Now press **X** and give the filename of your Pascal object file. It will be loaded, then the program will automatically write your file to disk as a command file.
11) When this is done, you will be returned to DOS and may now test your program. If your program was called ***CHESS/CMD***, you could run it by typing **CHESS** and pressing *ENTER*.

WARNING: Always test your command file from a cold start on a fresh disk before assuming it works.

Demonstration Programs.

Your diskette includes the utility programs TEXT/CMD and ASCII/CMD, a set of graphics and random number extensions source code procedures under the names GRAPHIC/SRC and RANDOM/SRC, and an additional procedure for print formatting under the filename GOTOXY/SRC. There are also four demonstration programs, CREATE/SRC, MAILIST/SRC, PRIME/SRC, and COINTOSS/SRC.

PRIME/SRC is a sample program that will find all prime numbers between 1 and 20,000 using the "Sieve of Erasthenes. It demonstrates the effective use of an array and shows the power of Pascal 80.

COINTOSS/SRC is a program created to demonstrate INCLUDE, RND(N), RNDR, PSET, PRESET, POINT, and GOTOXY. The program will continuously simulate flipping a coin ten times and graph the number of heads obtained on an axis from 1 to 10. When one bar of the graph (almost inevitably the bar for 5 Heads) reaches the top of the screen, the graph is rescaled.

The programs CREATE/SRC and MAILIST/SRC form a naive mailing list application meant as a demonstration of file handling techniques. You must first create a dummy file with CREATE, then you can run MAILIST to use the application. You have the following commands, which will be displayed at the bottom of the screen while the program is running;

T Get the first (Top) record.
+ Get the next record.
- Get the previous record.
L Get the Last record.
F Search for (Find) a last name.
N Add a New record.
D Delete current record.
H Print (Hardcopy) the current record.
P Print the full list.
Q Return (Quit) to the monitor or DOS.

This mailing list was intended only to demonstrate file handling, and no representation is made as to its suitability for any particular purpose. However, feel free to tailor it to your own applications.

COMPILATION ERROR MESSAGES

The compiler will stop when it finds an error, with an arrow pointing to the place where it discovered a problem. This may be in the line after the error (for example, if a semicolon is missing after a statement). A missing End may not be discovered for several lines.

BAD OPTION

Pascal-80 assumes that anything which you type before the keyword ***Program*** is an instruction to the compiler; see the section on *Compiler Options* .

SYNTAX ERROR

Something's wrong, but the compiler doesn't know what. This error may mean that you left out a semicolon between statements or tried to begin a number with a decimal point (Pascal requires **0.2** , not **.2**).

UNDECLARED

An identifier (usually a variable) or a label hasn't been declared; Pascal requires that you declare all variables (in a **var** statement) before you use them. Array index variables used in a record must be global variables in Pascal 80; Local variables will generate this message.

DUPLICATE

You declared a name twice in the same block. This may also mean that you used the same name in both a record and elsewhere in your program, as Pascal 80 will not allow this. Or you tried to declare a filename with the same name as one of the standard identifiers (Pascal does allow you to redefine these standard identifiers, but not as filenames.)

BAD RANGE

An array or subrange has been declared with an illogical range (such as 10..1).

REAL OVERFLOW

A real constant has a magnitude outside the permitted range of 1E-63 to 1E+63.

BAD TYPE

An illegal type declaration. Note, for example, that Pascal requires that parameters be of a predefined type: ***type Ran = 1..10; procedure Test (par:Ran)*** is OK, but not ***procedure Test (par:1..10)*** . Note also that Pascal-80 does not support certain structures, such as ***File of File***.

OUT OF MEM.

Usually means that the compiler has run out of symbol table space: you can use the *MEMORY* compiler option to keep track of how much space is left. This can also occur if the compiler runs out of space for storing labels (maximum of 63) or disk filespecs (maximum of 12). Also, a program may not contain more than 252 different scalar types, and a scalar may not have more than 255 elements. Occasionally a short but deeply-nested program may cause the compiler to run out of stack space before it runs out of symbol table space. If this happens, it will print *OUT OF MEM* , and then immediately assign more space to the stack (at the expense of the symbol table) and start the compilation over again. The additional stack space will continue to be available in future compilations until you reload the system.

MISMATCH

An attempt to perform an operation or assignment with elements of incompatible types (such as 'X'+2). Note that Pascal allows integers to be assigned to real variables, but (unlike BASIC) it does not allow real values to be assigned to integer variables (use the ***trunc*** function). Incidentally, -32768 is of type *real* under the rules of Pascal syntax (it's the negative of 32768, which is a real constant), and cannot therefore be assigned to an integer variable. A mismatch can also result from incompatible file operations, such as EOF(Output) or an attempt to reference a file name which has not been declared in a var statement (since the compiler knows the name only as a filespec, and not as a Pascal identifier).

UNRESOLVED GOTO

The destination of a goto statement doesn't exist in the program. It is always printed at the very end of compilation, since the compiler keeps hoping that the label will turn up...

STRUCTURE TOO BIG

An attempt to declare a set with more than 256 members (or with integers outside the range 0..255). Or an attempt to allocate more than 65535 bytes of storage to a block (usually in the form of arrays). Or a structure that is too deeply-nested for the compiler to handle (Array of Array of Array...to a depth of about 30). Or an attempt to pass an array (or record) with more than 510 bytes as a value parameter (this restriction does not apply to variable parameters).

BREAK

You stopped the compilation by holding down the *BREAK* key.

RUN-TIME ERROR MESSAGES

When a run-time error occurs, execution stops, and the system prints (in hex) the location where the error occurred. This location corresponds to the number printed at the left of each line during compilation, and allows you to find the approximate location of the error.

Some run-time errors do not need much explanation: **OUT OF MEM., DIV. BY 0, DISK ERROR, BEYOND EOF** . Less self-explanatory errors are:

BAD RANGE

A subscript of an array, or the value of a subrange-type variable is outside the range which you specified in your program.

REAL OVERFLOW

The result of a computation has a magnitude outside the range IE-64 <= N < IE+63 (this includes both underflow and overflow conditions).

INT. OVERFLOW

The result of a integer operation is outside the range -32768 <=n< =32767. Note that Pascal (unlike BASIC) does not automatically convert a result to *real* if it is too large.

MISMATCH

An invalid character was found while attempting to read a number from a disk file, or a non-integer was found when trying to read into an integer variable. If this happens when reading a number from the keyboard (file *INPUT*), the message **REDO** is printed.

STRUCTURE TOO BIG

An attempt to create a set at run-time with more than 256 elements: for example (.A,.B), if A=1 and B=300, or an attempt to assign a bigger set to a set variable than that variable was declared to have room for. Since space is allocated to sets in multiples of 16 elements, a set declared as 0..10 may actually accept elements up to 15 without an error.

ILLEGAL JUMP

It is not legal to jump (with *goto*) into a *for* loop or *case* statement or into an inactive procedure or function. In general, you can jump from a deeper nesting level to a shallower level (out of a for loop, for example), but you cannot jump deeper. If you jump out of a function without assigning a value to the function, the value 0 will be returned. Pascal 80 will allow you to jump from inside one for loop to another one at the same nesting level, but the value of the control variable will be undefined.

Upgrade History

Pascal 80 was originally released by Ramware, a division of Soft-Side magazine, in early 1981. The original version worked only on the TRS-80 Model I under TRS-DOS 2.3, and was reviewed in the December 1981 issue of Byte magazine.

In early 1982, the program was extensively revised by New Classics Software. New features added the Include faciity from UCSD Pascal, compatibility with all TRS-DOS replaement operating systems on both the model I and the model III, lower case support in printed output, utilities to convert files between packed and ASCII formats, a new editor with character insert and delete, protected memory for machine language programs, and a new manual. This version was released at the West Coast Computer Faire on March 22, 1982.

Three bugs were found in the new version. There was a conflict between CLS and file handling that caused files to crash after CLS was executed. The Seek command failed to read into the buffer after finding information. The ZERO compiler option zeroed all high memory, including protected memory. These problems were fixed, and two enhancements were made. The first enhancement was a modification to the RECORD command to allow declarations like SEX: (Male,Female) within a record. This introduced a new problem that still exists in the program. Record field names are no longer completely local to the Record, and cannot be duplicated in other parts of the program. The second enhancement was the addition of extra TRS-80 graphics routines and random number functions.

These changes became revision A, released on March 28, 1982.

One more bug was found later. The utility program TEXT/CMD was found to overflow the stack and crash. Both TEXT and ASCII were changed to create their own stack instead of using the DOS stack. Then further enhancements were made. Then two new functions were added to the editor, for block moves and printing source programs, to form Revision C, released April 24, 1982.

Revision D, Released June 1, 1982, added support for lower case in compiler options and changed the cursor choice option routines.

We would also like to either support pointer variables, or at least simulate them so that schools could teach the use of pointers using Pascal 80. Any user suggestions, or better yet, completed modifications for Pascal 80, are eagerly solicited.

Using Pascal 80 to Teach Programming

Pascal was originally designed as a language to teach computer programming. Pascal 80 updates standard Pascal from the card reader and teletype era to the age of keyboards and video displays. Pascal 80 was written, revised, and documented by professional educators specifically for use in teaching computer literacy in high schools, technical schools, and colleges.

Currently, many schools are using Basic to teach programming. This is largely a result of the limited memory that was available on early microcomputers. Since a general purpose Basic could be put in 4K of ROM and used with 4K of RAM, most early microcomputers came with a Basic interpreter. Pascal requires significantly more computer resources. However, Basic has many disadvantages as a teaching language. Lacking a choice of control structures similar to Pascal's **while ... do** and **repeat ... until**, Basic is not suited to top down programming. This frequently leads to abuse of GOTO statements and other bad habits that can actually hinder future success in programming. The line numbers in Basic reduce readability and introduce added complexity to the language. The lack of support for pretty printing in most Basic implementations results in programs even harder to understand. The lack of named procedures and functions, as well as the limitations on significant characters in variable names, also create obstacles to learning. Basic lacks the sophistication of modern programming languages, missing features such as user defined types, structured variables and dynamic pointers. For these reasons, schools should probably teach Pascal instead of Basic, not as a second or optional language.

One of the major obstacles in learning a compiled language is the time delay introduced by disk access. An illustration of the delay is the process of writing a short program on the IBM personal computer with IBM Pascal. After loading a text editor and writing the program, the programmer saves it as a disk file, then inserts and runs the first pass compiler. This requires answering several questions. Then this disk is removed and the second pass compiler is inserted and executed. Then this disk is removed and the disk with the Linker is inserted and run, requiring the programmer to answer several more questions. Then the student can switch back to the second drive and run the program. Altogether, it takes 5 disks, the execution of 5 programs, answering a dozen questions, and the creation of 6 disk files to get

even a tiny program to run. Students should not have to suffer through this. They should receive immediate correction when they make a mistake and immediate reinforcement when they do well. Pascal 80 was written to compile and run without disk access in order to allow students to spend their class time writing programs and correcting their errors, not saving and executing disk files. In Pascal 80, plain English error messages, with an arrow pointing to the location where the error was discovered, provide immediate correction. Since compilation stops when an error is discovered, students deal with one error at a time. In seconds, they can return to the editor, (coming automatically to the spot where the error was located) fix the error, and return to compilation.

Persons teaching Advanced Placement Pascal on the high school level will want to use Pascal 80 for at least the first half of the course, and then switch to a different version of the language so that students learn to cope with more primitive editors and more involved disk access. Pascal 80 also does not include pointer variables, which are mandated by the colleges for Advanced Placement Pascal courses. It is suggested that the following commands and topics be reserved for the last part of the course, and taught using another version of Pascal;

Pointer variables, **new** and **dispose**
Variant records, **with**
get and **put**

Particular attention should be given in an introductory course in programming to portability, modularity, meaningful names for procedures, functions, and variables, stepwise refinement, top-down and bottom-up programming, localization and the avoidance of side effects, testing of small parts of a program with boundary conditions, and the elements of pretty printing, including use of lower case, comments, and the identification of blocks by matching indentation.

The extensions in Pascal 80, including seek, cls, peek, poke, inkey, fp, ex, real variables, and the gotoxy, random, and graphics routines should be avoided in class. Standard Pascal does not have random access files, so instruction should concentrate on text files. Some teachers may wish to deny students access to this manual in order to keep non standard functions from appearing in student programs. If file handling is taught with Pascal 80, the students will have to learn to use close and non standard uses of read and write.

Reviews of Books on Pascal

The following books were consulted in preparing this manual and in teaching Pascal 80 to high school computer science teachers. The books are listed in order of my own estimate of their value.

By far the most helpful book consulted was **Oh! Pascal!** by Doug Cooper and Michael Clancy of the University of California at Berkeley. The book is well written, has clear explanations, and great care has been taken in the layout of the book to insert examples at the right place in the text and highlight important features. The authors see programming as a separate discipline, not as a branch of mathematics, so mathematical examples are kept to a minimum. Every chapter has a section on programming technique, called *Antibugging and Debugging*. These sections are extremely valuable. The program examples are presented in upper and lower case, and are very readable. Many example programs, both short and long, are given. There are self test questions for each chapter, with answers in the back of the book, as well as additional exercises without answers given. Many of the exercises are quite demanding, and the reading level is fairly high, probably beyond high school students. The use of Pascal in an interactive environment is taken for granted, making the book exceptionally useful with microcomputers. The book sells for $16, and is published by Norton.

Programming in Pascal by Peter Grogono, seems to be the standard college textbook on Pascal. It is clearly written and well organized, and is organized around a number of large sample programs. Grogono concentrates on the original mainframe implementation of Pascal. This is the book recommended by Phelps Gates, the author of Pascal 80.

Pascal User Manual and Report, by Kathleen Jensen and Niklaus Wirth, is the original sourcebook and official bible of Pascal. It gives a clear and brief explanation of the features of Standard Pascal, and is a good book for experienced programmers who wish to learn the language quickly. However, as the title indicates, it is a report on the language, not a tutorial.

Introduction to Pascal by Rodnay Zaks is a competent, clearly written tutorial, though unexciting. It does tend to confuse the reader by mixing material on UCSD Pascal in the same chapters as Standard Pascal, but UCSD material is clearly identified for those who wish to learn Standard Pascal first. Although the programs are in upper case only, bold face is used to make Pascal keywords stand out and the book is well laid out, so readability does not suffer.

While this book is still at an elevated reading level, an average (but not below average) high school sophmore should be able to understand it. If I had to select one of the books listed here to teach programming in high school, I would reluctantly choose this text, though I would want to use the supplementary material and some of the exercises from Oh! Pascal! as well. The publisher is Sybex, the price $14.95

Pascal, An Introduction to Methodical Programming, by W. Findlay and D.A. Watt of the University of Glasgow, is a college text. The programs are given in upper case only, making them difficult to read. There are also a lot of irritating forward and backward references to program examples. While this book is much less valuable than Oh! Pascal!, it may be useful to those wishing a second reference to clarify points they don't understand from the other book. The section on Pointers is particularly good. Emphasis is placed on syntax diagrams, which are fairly well done. Like Oh! Pascal!, this text is not mathematically oriented. The publisher is Computer Science Press in Rockville, Maryland.

Foundations of Programming with Pascal, by Lawrie Moore of the University of London, is an expensive (about $50) and complex text with long sections on number systems, base conversion, electrical examples of logic gates, and digressions into topics such as Backus Naur form and Venn Diagrams. It is more mathematically oriented than the other books, and even includes a chapter on *Using an Efficient Method of Integration.*

Program examples are given in upper and lower case, and are quite readable. This is a text for well educated, mathematically oriented academics who also want a grounding in the jargon of the high priesthood of computer science.

For those interested in moving on to UCSD Pascal, the book by Rodnay Zaks above will be a good start. Another useful book is **Beginner's Guide to the UCSD Pascal System** by Kenneth Bowles, from Byte Books. The material is decent, although the layout of the book is terrible, the programs are in all upper case, and many of the illustrations are mediocre black and white photographs of video display screens.

Those with Apple computers will find **Apple Pascal, a hands on approach** , by Arthur Luermann and Herbert Peckham, a useful orientation to Apple Pascal, after they learn standard Pascal. This is not a good book for classroom use, as it concentrates too much on the specific features of Apple Pascal to the confusion of Standard Pascal.

In every race, someone has to finish last. The **Pascal Primer** by David Fox and Mitchell Waite, tries to be humorous and cute, but only succeeds in being glib and superficial. The book is not all bad, and is often lucid and readable, but the other texts listed above do a better job. The Pascal Primer concentrates on UCSD Pascal, with special emphasis on the Apple.

I also have **The Pascal Handbook** by Jacques Tiberghien, published by Sybex. When I received the book, it seemed a good idea to have a reference listing all the Pascal Commands with syntax diagrams and sample programs. Despite the fact that it sits on a shelf within reach of my Apple and both TRS-80s, I never reach for it. The few times I tried to use it, months ago, it just didn't have the information I needed. I suggest using the index in Oh! Pascal! and looking up commands that way.

Chaining program files under Pascal 80

We cannot guarantee that it will work with all files. In fact, we have had problems with single character file names. However, we have had success with it in continuous chaining of object files stored on diskette.

This procedure is on your disk as CHAIN/SRC:

```
(* This module requires the following type declaration
   at the beginning of the program:
   type Filename = array [1..23] of char;)
procedure Chain (a:filename);
var i,j : integer;
   good : set of char;
begin
   good := ['A'..'Z']+['/','.', ':']+['a'..'z']+['0'..'9'];
   i := 21248;
      (* 5300H = DCB *)
   j := 1;
   while ((a[j] in good) and (j<24)) do
      begin
      if a[j] in ['a'..'z'] then poke(i,ord(a[j])-32)  else poke(i,ord(a[j]));
      i := i+1; j := j+1;
      end;
   poke(i,ord('$'));
   i := 22206; poke(i,24); i := i+1;   (* no time *)
   poke(i,13);
   i := 28096; poke(i,62); i := i+1;   (* patch end *)
   poke(i,1); i := i+1; poke(i,50);
   i:= i+1; poke(i,86); i := i+1;
   poke(i,112); i := i+1; poke(i,24);
   i := i+1; poke(i,12);
   i := 28393; poke(i,125);          (* patch end *)
   i := 28247; poke(i,195); i := i+1   (* patch abort *)
   poke(i,232); i := i+1; poke(i,110);
   i := 31724; poke(i,62);          (* X title *)
   i := i+1; poke(i,124);
   i := call(31723,0)
end;
```

Here is a sample program that uses the CHAIN procedure correctly:

```
program Test;
type filename = array[1..23] of char;
(*$ CHAIN/SRC *)
begin
  writeln('Which program do you wish to run (A/B/C) ?');
  repeat c := inkey until ord(c) <> 0;
  case c of 'A': chain('PROGA/OBJ');
            'B': chain('PROGB/OBJ');
            'C': chain('PROGC/OBJ');
       end
end.
```

Only p-code files created by Pascal 80's Write command may be called as chain files. However, the calling program may be a text program or a command program created by the author package. Programs called by chain can themselves contain further chain commands. Therefore, if you are creating a turnkey application with Pascal 80, create a startup command file and set AUTO to execute it. Store the remainder of your files as Pascal 80 object files and call them as needed from each other.

Control Key Problems

On certain early Model I and Model III computers, the ROM does not implement the shift down arrow as a control key. Since this makes it impossible to use the editor, it is necessary to supply a separate keyboard driver. Some disk operating systems, including LDOS and DOS-Plus, have keyboard drivers that work. The disk also contains a Model III keyboard driver under the filename **CTRLKEY/CMD**. To use it, type **CTRLKEY** *ENTER* from DOS before typing PASCAL. It will replace the other keyboard driver in memory. This may disable some functions such as the JKL screen print routine in NewDOS.

Model III users who wish to use the new keyboard driver can construct a BUILD file to load the driver automatically and then call Pascal. To do this, from DOS, type:

BUILD START *ENTER*

when you get the prompt, type:

CTRLKEY *ENTER*

at the next prompt, type:

PASCAL *ENTER*

at the next prompt, press *BREAK*.
When you return to S, type:

AUTO DO START *ENTER*

Now, all you will have to do to load the driver and Pascal is press the reset key.

The control key routine loads into memory locations FEAO to FFFF, and therefore requires a 48K Model III computer.

Control Key Problems

On certain early Model I and Model III computers, the ROM does not implement the shift down arrow as a control key. Since this makes it impossible to use the editor, it is necessary to supply a separate keyboard driver. Some disk operating systems, including LDOS and DOS-Plus, have keyboard drivers that work. The disk also contains a Model III keyboard driver under the filename **CTRLKEY/CMD**. To use it, type **CTRLKEY** *ENTER* from DOS before typing PASCAL. It will replace the other keyboard driver in memory. This may disable some functions such as the JKL screen print routine in NewDOS.

Model III users who wish to use the new keyboard driver can construct a BUILD file to load the driver automatically and then call Pascal. To do this, from DOS, type:

BUILD START *ENTER*

when you get the prompt, type:

CTRLKEY *ENTER*

at the next prompt, type:

PASCAL *ENTER*

at the next prompt, press *BREAK*.
When you return to S, type:

AUTO DO START *ENTER*

Now, all you will have to do to load the driver and Pascal is press the reset key.

The control key routine loads into memory locations FEAO to FFFF, and therefore requires a 48K Model III computer.

Table of Contents